

# Network Coded TCP (CTCP)

MinJi Kim\*, Jason Cloud\*, Ali ParandehGheibi\*, Leonardo Urbina\*,  
Kerim Fouli\*, Douglas Leith†, Muriel Médard\*

## ABSTRACT

We introduce CTCP, a reliable transport protocol using network coding. CTCP is designed to incorporate TCP features such as congestion control and reliability while improving on TCP's performance in lossy and/or dynamic networks. CTCP builds upon the ideas of TCP/NC introduced by Sundararajan et al. and uses network coding to provide robustness against losses. We provide an implementation of CTCP (in userspace) and demonstrate its performance in both testbed and production networks.

## 1. INTRODUCTION

The Transmission Control Protocol (TCP) is one of the core protocols of today's Internet Protocol Suite. TCP was designed for reliable transmission over wired networks, in which losses are generally an indication of congestion. This is not the case in wireless networks, where losses are often due to fading, interference, and other physical phenomena. While there has been considerable work on improving the operation of TCP over wireless networks, particularly in WiFi networks, TCP still performs poorly in many current systems. Approaches tailored to specific wireless systems may perform very well over the networks for which they have been designed but may have limited applicability as users move among different WiFi network instances, or operate on both WiFi and cellular networks. In general, the performance of wireless networks is highly variable across different deployments [1] and even within a network according to whether the devices are hand-held or not [2]. Users may also experience losses with wireline networks, for instance over DSL connections [3, 4].

The goal of our design is to keep traditional TCP's best features, including congestion control and reliability, and augment them with network coding for resilience against losses and failures in the network without explicit consideration of the lower layers. We do not adopt a clean slate or physical-layer dependent approach that would incorporate physical layer or link

layer methods to reduce the types of losses that lead to poor performance of TCP. Such methods would generally require customization, whereas we seek an approach that is widely applicable to different lossy settings. Indeed, the measurements we present in our work show that we may more than double throughput with our approach over a variety of lossy networks, such as a testbed with controlled losses, a WiMAX system and a sample of public WiFi systems, where we have no access to the routers. Our approach is thus deliberately not transformative but simply meliorative - we seek a technique that can be retro-fitted to a variety of existing networks as well as bringing benefits in future networks.

In order to combine the benefits of TCP and network coding [5, 6], Sundararajan et al. in [7] proposed a new protocol called TCP/NC. Packets in TCP/NC use random network coding [8] to single flows. TCP/NC modifies TCP's acknowledgment (ACK) scheme so that it acknowledges *degrees of freedom* (dofs) instead of individual packets. This is done by using the concept of seen packets. Packets are acknowledged in order, possibly before they are decoded, but not before contributing to the construction of received packets. Such an approach implies that the number of innovative coded packets received, which we term degrees of freedom, is translated to the number of consecutive packets received, even before packets may have been decoded. Reference [9] provides a simple mathematical model and analysis with simulation results that show that TCP/NC achieves significantly higher throughput than traditional versions of TCP in lossy networks.

Our protocol, CTCP, builds upon TCP/NC introduced by [7]. However, we enhance the algorithms from TCP/NC to make CTCP more efficient and robust. Therefore, CTCP departs from TCP/NC in the following ways.

1. We implement the design in C for Linux operating systems. We implement the protocol in userspace (over UDP) for ease of implementation and modifications. The implementation demonstrates that CTCP is indeed able to achieve high throughput despite losses in the network.
2. CTCP, unlike TCP/NC which introduces network coding indirectly through a shim layer between TCP and IP layers, is a transport protocol that

\*M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, and M. Médard are with the Massachusetts Institute of Technology, MA USA (e-mail: {minjikim, jcloud, parandeh, lurbina, fouli, medard}@mit.edu).

†D. Leith is with the Hamilton Institute, NUI Maynooth, Ireland (e-mail: doug.leith@nuim.ie).

uses network coding directly. At a first glance, this may not be a significant change; however, by designing a new transport protocol, we are able to leverage network coding more efficiently and introduce congestion control mechanisms better suited to a protocol using network coding.

3. CTCP is adaptive. TCP/NC assumes a known average end-to-end packet loss rate  $p$  and determines a redundancy factor  $R \sim \frac{1}{1-p}$  for the communication [7]. Note that this redundancy factor plays a key role in enabling TCP/NC to use network coding to overcome losses. However, in a real network,  $p$  is rarely known a priori and fluctuates over time and space. Our protocol estimates  $p$  and dynamically adjusts the redundancy rate  $R$  to adjust to the losses on the fly.
4. CTCP uses systematic block coding to manage delay and complexity. TCP/NC uses a sliding window approach for coding operations, which can have significant decoding delay at the receiver as it may have undesirable worst-case behavior. Furthermore, the use of a systematic code significantly reduces the decoding overhead over random solutions with dense matrices. When there are no losses ( $p = 0$  leading to  $R \sim 1$ ), CTCP, in effect, reduces to a TCP-like protocol without coding.
5. We provide a congestion control mechanism that works well with network coding. Traditional TCP uses a sliding transmission window with sequence numbers to identify bytes. With coding, any coded packet within a block can replace another packet; therefore, we modify the congestion control mechanism to use *tokens* – i.e. a token allows CTCP sender to transmit a packet. Our congestion control mechanism generates or destroys tokens to adjust CTCP sender’s transmission rate.

This paper discusses the algorithmic and implementation details of CTCP and provides extensive experimentation results to verify its performance. The rest of the paper is organized as follows. In Section 2, we present some related material. In Section 3, we provide an overview of CTCP. In Sections 4 and 5, we describe CTCP sender and receiver, respectively. After describing the protocol, in Sections 6 and 7, we demonstrate CTCP’s performance and compare it with traditional TCP variants in a testbed as well as in real-world production networks, such as WiMAX and publicly available WiFi networks.

## 2. RELATED WORK

In order to mitigate the effect of losses on TCP, particularly in the context of wireless links, the use of coding has been proposed before our work or that of [7]. We may roughly taxonomize these into approaches that

consider the operation of TCP jointly with lower layer redundancy, at the physical or MAC level, often in a cross-layer approach [10–18] and those that consider redundancy at the IP layer or above without access to the lower layers [19–25]. Since our goal is to provide a system that operates without access to, or even detailed knowledge of, the physical layer, it is in the latter category that our approach belongs, although we shall address some issues related to lower layer reliability mechanisms, such as hybrid ARQ (HARQ), in the latter part of the paper, when discussing experiments over WiMAX systems.

The approaches in papers implementing coding at the IP layer for operation with TCP have generally revolved around traditional block codes, often Reed-Solomon codes. Sometimes these approaches are combined with mechanisms such as explicit congestion notification (ECN) [16, 20]. These block-based approaches acknowledge packets upon successful decoding and, if the number of errors exceeds the predetermined level of redundancy, a decoding failure occurs, requiring retransmission of the block. Even though many of the approaches at the IP layer or above, often, like the work in this paper, consider adaptation to the observed losses [18, 24], the mechanism remains one in which acknowledgements require decoding of full blocks and reception of a full block must occur before the window can progress. The concept of ACK for seen packets, mentioned in the introduction, circumvents the need for decoding before acknowledging.

Moreover, the use of structured codes, in block format or otherwise, is not necessary to combat erasures. Random network coding [8] can be used to combat random erasures over networks in unicast and multicast settings to obtain the optimal rate without any need to know where and when erasures occur within the network [26]. The use of random network coding has been demonstrated successfully for broadcast erasure correction in wireless networks [27]. The concept of random coding with seen-style ACKs has also been proposed for delay-tolerant networks (DTNs) in [28].

## 3. OVERVIEW OF CODED TCP (CTCP)

Before discussing the CTCP sender and receiver in detail, we provide a more holistic view of the two here.

The CTCP sender segments the stream, or the file, into a series of blocks as shown in Figure 1. A block is chosen to be of a fixed size, equivalent to *blksize* number of packets where each packet is assumed to be of fixed length. If the remainder of a file or a stream is not large enough to form a complete packet, the packet is padded with zeros to ensure that all packets are of the same length. A block need not be completely full, i.e. a block may have fewer than *blksize* packets; however, block  $i$  should be full before block  $i + 1$  is initialized.

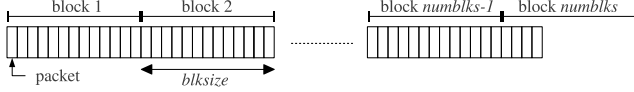


Figure 1: CTCP sender divides the data into blocks.

The CTCP sender keep  $numblks$  of blocks in memory and the value of  $numblks$  should be conveyed to the receiver. The value of  $numblks$  may be negotiated at initialization between the sender and the receiver, as  $numblks$  directly affects the memory usage on both ends. We denote the smallest block in memory to be  $currbk$ . Note that this does not mean that CTCP sender may send  $numblks \times blksize$  amount of data at any given point in time.

The sender is allowed to transmit packets only if the congestion control mechanism allows it to; however, whenever it is allowed to transmit, the sender may choose to transmit a packet from any one of the blocks in memory, i.e. blocks  $currbk$ ,  $currbk + 1$ , ...,  $currbk + numblks - 1$ . In Section 4.5, we shall discuss the sender's algorithm for selecting a block from which it sends a packet. The payload of the transmitted packet may be coded or uncoded; the details of the coding operations can be found in Section 4.4.

The sender includes the following in each packet:

- the block number,
- a seed for a pseudo-random number generator, which allows the receiver to generate the coding coefficients,
- the sequence number  $seqno$ , and
- the (coded or uncoded) payload.

The sequence number for CTCP differs from that of TCP – for TCP, a sequence number indicates a specific data byte; for CTCP, a sequence number indicates that a packet is the  $seqno$ -th packet transmitted by the sender, thus, is not tied to a byte in the file.

The CTCP receiver sends acknowledgments (ACKs) for the packets it receives. In the ACK, the receiver indicates

- the smallest undecoded block  $ack\_currbk$ ,
- the number of degrees of freedom (dofs)  $ack\_currdof$  it has received for the current block  $ack\_currbk$ , and
- the  $ack\_seqno$  of the packet it is acknowledging.

Using the information in an ACK, CTCP sender adjusts its behavior. We first describe the sender algorithm in Section 4, as most of the intelligence of the protocol is on the sender's side. Then, we present CTCP receiver algorithm in Section 5. CTCP receiver's main role is to decode and deliver data to the application.

Table 1: Definitions of the sender parameters

Notation	Definition
$p$	Short term average packet loss rate
$RTO$	Retransmission timeout period (equal to $\gamma \cdot RTT$ where $\gamma \geq 1$ is a constant)
$RTT$	Current (or the last acknowledged packet's) round-trip time
$RTT_{min}$	The minimum round-trip time
$seqno\_nxt$	The sequence number of the next packet to be transmitted
$seqno\_una$	The sequence number of the latest unacknowledged packet
$ss\_threshold$	Slow-start threshold, i.e. if $tokens > ss\_threshold$ , the sender leaves the slow-start mode
$time\_lastack$	Timestamp of when the sender received the latest ACK (initialized to the time when the sender receives a SYN packet from the receiver)
$tokens$	Number of tokens, which is conceptually similar to congestion window for traditional TCP
$blksize$	Size of the blocks (in number of packets)
$currbk$	Current block number at the sender, which is the smallest unacknowledged block number
$currdof$	Number of dofs the receiver has acknowledged for the current block
$numblks$	Number of active blocks, i.e. the sender may schedule and transmit packets from blocks $currbk$ , $currbk+1$ , ..., $currbk+numblks-1$
$B(seqno)$	Block number from which packet with $seqno$ was generated from
$T(seqno)$	Timestamp of when packet with $seqno$ was sent

## 4. CTCP SENDER

We present the sender side algorithm for CTCP. The sender maintains several internal parameters (defined in Table 1), which it uses to generate coded packets and schedule blocks to be transmitted.

### 4.1 Network Parameter Estimation

The CTCP sender estimates the network parameters, such as  $RTT$  and  $p$ , using the ACKs as shown in Algorithm 1. The sender adjusts its actions, including coding operations and congestion control depending on these estimates. If the sender does not receive an ACK from the receiver for an extended period of time (i.e. time-out occurs), the network parameters are reset to

```

Receive an ACK;
time_lastack ← current time;
RTT ← time_lastack − T(ack_seqno);
RTT_min ← minimum of {RTT, RTT_min};
if ack_currblk > currblk then
    Free blocks currblk, ..., ack_currblk − 1;
    currblk ← ack_currblk;
    currdof ← ack_currdof;
end
currdof ← max{ack_currdof, currdof};
if ack_seqno > seqno_una then
    losses ← ack_seqno − seqno_una + 1;
    p ← p(1 − μ)losses+1 + (1 − (1 − μ)losses);
end
seqno_una ← ack_seqno + 1;

```

**Algorithm 1:** CTCP sender algorithm for updating the network parameters. Above,  $\mu$  is the smoothing factor used for our modified exponential smoothing technique.

predefined default values. The predefined default values may need to be chosen with some care such that they estimate roughly what the network may look like.

The CTCP sender maintains moving averages of  $p$ . We use a slightly modified version of the exponential smoothing technique (for reasons discussed later in this paragraph). For  $p$ , we can consider the data series we are averaging to be a 0-1 sequence, where 0 indicates that the packet has been sent successfully and 1 otherwise (i.e. a packet loss). Now, assume that there were  $losses$  number of packets lost. If  $losses = 0$ , then the update equation for  $p$  in Algorithm 1 becomes

$$p \leftarrow p(1 - \mu) + 0, \quad (1)$$

where  $\mu$  is the smoothing factor. If  $losses = 1$ , the same update equation becomes

$$p \leftarrow p(1 - \mu)^2 + \mu = (1 - \mu)[p(1 - \mu) + 0] + \mu, \quad (2)$$

which is identical to executing an exponential smoothing over two data points (one lost and one acknowledged). We can repeat this idea for  $losses > 1$  to obtain the update rule for  $p$  in Algorithm 1. Therefore, the fact that a single ACK may represent multiple losses ( $losses \geq 1$ ) leads to a slightly more complicated update rule for  $p$  than that for  $RTT$  as shown in Algorithm 1. To the best of our knowledge, such an update rule has not been used previously.

## 4.2 Reliability

CTCP achieves reliability by ensuring that each block is received and decoded. In Algorithm 1, CTCP sender increments  $currblk$  only if it has received an ACK indicating that the receiver is able to decode  $currblk$  – i.e.

$ack\_currblk > currblk$ . This mechanism is equivalent to traditional TCP’s window sliding scheme in which the TCP sender only slides its window when it receives an ACK indicating the some bytes have been received. In the case of CTCP, the reliability is implemented over blocks instead of bytes.

## 4.3 Congestion Control Mechanism

Traditional TCP’s AIMD congestion control increases TCP sender’s congestion window size  $cwnd$  by  $\alpha$  packets per RTT and multiplicatively decreases  $cwnd$  by a backoff factor  $\beta$  on detecting packet losses within one RTT inducing a single  $cwnd$  backoff. The usual values are  $\alpha = 1$  when appropriate byte counting is used, and  $\beta = 0.5$ . On lossy links, repeated backoffs in response to noise losses rather than queue overflow can prevent  $cwnd$  from increasing to fill the available link capacity. The behavior is well known and is captured, for example, in [29] in which  $cwnd$  scales as  $\sqrt{1.5/p}$ , where  $p$  is the packet loss rate.

The basic issue here is that on lossy links, loss is not a reliable indicator of network congestion. One option might be to use delay, rather than loss, as the indicator of congestion, but this raises many new issues and purely delay-based congestion control approaches have not been widely adopted in the internet despite being the subject of extensive study. Another option might be to use explicit signalling, for example via ECN, but this requires both network-wide changes and disabling of  $cwnd$  backoff on packet loss. These considerations motivate consideration of hybrid approaches, making use of both loss and delay information. The use of hybrid approaches is well-established, for example Compound TCP [30] is widely deployed.

We consider modifying the AIMD multiplicative backoff. Before discussing the details of backoff behavior, we emphasize that CTCP uses *tokens* to control the CTCP sender’s transmission rate instead of the congestion window  $cwnd$ ; therefore, *tokens* play a similar role for CTCP as  $cwnd$  does for TCP. A token allows the CTCP sender to transmit a packet (coded or uncoded). When the sender transmits a packet, the token is used. The number of tokens, *tokens*, is controlled according to the modified AIMD multiplicative backoff (Algorithm 2).

As shown in Algorithm 2, we modify the AIMD multiplicative backoff to have

$$\beta = \frac{RTT_{min}}{RTT}, \quad (3)$$

where  $RTT_{min}$  is the path round-trip propagation delay (which is typically estimated as the lowest per packet RTT observed during the lifetime of a connection) and  $RTT$  is the current round-trip time.

This is similar to the approach considered in [31], which uses  $\beta = RTT_{min}/RTT_{max}$  with the aim of mak-

```

if current time > time_lastack + RTO then
  | tokens ← initial token number;
  | Set to slow-start mode;
end
if Receive an ACK on path i then
  | if slow-start mode then
  | | tokens ← tokens + 1;
  | | if tokens > ss_threshold then
  | | | Set to congestion avoidance mode;
  | | end
  | else
  | | if ack_seqno > seqno_una then
  | | | tokens ←  $\frac{RTT_{min}}{RTT} tokens$ ;
  | | else
  | | | tokens ← tokens +  $\frac{1}{tokens}$ ;
  | | end
  | end
end

```

**Algorithm 2:** CTCP sender algorithm for congestion control mechanism.

ing TCP throughput performance less sensitive to the level of queue provisioning. Indeed on links with only queue overflow losses, (3) reduces to the approach in [31] since  $RTT = RTT_{max}$  (the link queue is full) when loss occurs. In this case, when a link is provisioned with a bandwidth-delay product of buffering, as per standard guidelines, then  $RTT_{max} = 2RTT_{min}$  and  $\beta = 0.5$ , i.e. the behavior is identical to that of standard TCP. More generally, when queue overflow occurs the sum of the flows' throughputs must equal the link capacity  $B$ ,  $\sum_{i=1}^n tokens_i / RTT_i = B$  where  $n$  is the number of flows. After backoff according to (3), when the queue empties then the sum-throughput becomes  $\sum_{i=1}^n \beta_i tokens_i / RTT_{min,i} = B$ . That is, the choice (3) for  $\beta$  decreases the flow's *tokens* so that the link queue just empties and full throughput is maintained.

On lossy links (with losses in addition to queue overflow losses), use of  $RTT$  in (3) adapts  $\beta$  to each loss event. When a network path is under-utilized,  $RTT = RTT_{min}$  (therefore,  $\beta = 1$  and  $\beta \times tokens = tokens$ ). Thus, *tokens* is not decreased on packet loss. Hence, *tokens* is able to grow, despite the presence of packet loss. Once the link starts to experience queueing delay then  $RTT > RTT_{min}$  and  $\beta < 1$ , i.e. *tokens* is decreased on loss. Since the link queue is filling, the sum-throughput before loss is  $\sum_{i=1}^n tokens_i / RTT_i = B$ . After decreasing *tokens*, when the queue empties the sum-throughput is at least (when all flows backoff their *tokens*)  $\sum_{i=1}^n \beta_i tokens_i / RTT_{min,i} = B$ . That is, (3) adapts  $\beta$  to maintain full throughput.

Although we focus on using (3) in combination with linear additive increase (where  $\alpha$  is constant), we note that this adaptive backoff approach can also be com-

bined with other types of additive increase including, in particular, those used in Cubic TCP and Compound TCP. As shown here, these existing approaches can be extended to improve performance on lossy links.

#### 4.3.1 Mathematical Modeling

In this section, we provide some analysis on our choice of  $\beta$ , the multiplicative backoff factor.

Consider a link shared by  $n$  flows. Let  $B$  denote the capacity of the link and  $T_i$  the round-trip propagation delay of flow  $i$ . We will assume that the queueing delay can be neglected, i.e. the queues are small or the link is sufficiently lossy that the queue does not greatly fill. We also assume that any differences in the times when flows detect packet loss (due to the differences in path propagation delay) can be neglected. Let  $t_k$  denote the time of the  $k$ -th network backoff event, where a network backoff event is defined to occur when one or more flows reduce their *tokens*. Let  $w_i(k)$  denote the *tokens* of flow  $i$  immediately before the  $k$ -th network backoff event and  $s_i(k) = w_i(k) / T_i$  the corresponding throughput. With AIMD we have

$$s_i(k) = \tilde{\beta}_i(k-1)s_i(k-1) + \tilde{\alpha}_i T(k) \quad (4)$$

where  $\tilde{\alpha}_i = \alpha / T_i^2$ ,  $\alpha$  the AIMD increase rate in packets per RTT,  $T(k)$  is the time in seconds between the  $k-1$  and  $k$ -th backoff events,  $\tilde{\beta}_i(k)$  is the backoff factor of flow  $i$  at event  $k$ . The backoff factor  $\tilde{\beta}_i(k)$  is a random variable, which takes the value 1 when flow  $i$  does not experience a loss at network event  $k$ , and otherwise takes the value given by (3). The time  $T(k)$  is also a random variable, with distribution determined by the packet loss process and typically coupled to the flow rates  $s_i(k)$ ,  $i = 1, \dots, n$ .

For example, associate a random variable  $\delta_j$  with packet  $j$ , where  $\delta_j = 1$  when packet  $j$  is erased and 0 otherwise. Assume the  $\delta_j$  are i.i.d with erasure probability  $p$ . Then  $Prob(T(k) \leq t) = 1 - (1-p)^{N_t(k)}$  where  $N_t(k) = \sum_{i=1}^n N_{f,i}(t)$  is the total number of packets transmitted over the link in interval  $t$  following backoff event  $k-1$  and  $N_{t,i}(k) = \tilde{\beta}_i(k-1)s_i(k-1)t + 0.5\tilde{\alpha}_i t^2$  is the number of packets transmitted by flow  $i$  in this interval  $t$ . Also, the probability  $\gamma_i(k) := Prob(\tilde{\beta}_i(k) = 1)$  that flow  $i$  does not back off at the  $k$ -th network backoff event is the probability that it does see any loss during the RTT interval  $[T(k), T(k) + T_i]$ , which can be approximated by  $\gamma_i(k) = (1-p)^{s_i(k)T_i}$  on a link with sufficiently many flows.

Since both  $\tilde{\beta}_i(k)$  and  $T(k)$  are coupled to the flow rates  $s_i(k)$ ,  $i = 1, \dots, n$ , analysis of the network dynamics is generally challenging. However, when the backoff factor  $\tilde{\beta}_i(k)$  is stochastically independent of the flow rate  $s_i(k)$  then analysis is relatively straightforward. Note that this assumption is valid in a number of useful and interesting circumstances. One such cir-

cumstance is when links are loss-free (with only queue overflow losses) [32]. Another is on links with many flows and i.i.d packet losses, where the contribution of a single flow  $i$  to the queue occupancy (and so to  $RTT$  in (3)) is small. Further, as we will see later, experimental measurements indicate that analysis using the assumption of independence accurately predicts performance over a range of other network conditions, and so results are relatively insensitive to this assumption.

Given independence, from (4),

$$\mathbb{E}[s_i(k)] = \mathbb{E}[\tilde{\beta}_i(k)]\mathbb{E}[s_i(k-1)] + \tilde{\alpha}_i\mathbb{E}[T(k)]. \quad (5)$$

When the network is also ergodic, a stationary distribution of flow rates exists. Let  $\mathbb{E}[s_i]$  denote the mean stationary rate of flow  $i$ . From (5) we have

$$\mathbb{E}[s_i] = \frac{\tilde{\alpha}_i}{1 - \mathbb{E}[\tilde{\beta}_i]}\mathbb{E}[T]. \quad (6)$$

Since the factor  $\mathbb{E}[T]$  is common to all flows, the fraction of link capacity obtained by flow  $i$  is determined by  $\tilde{\alpha}_i/(1 - \mathbb{E}[\tilde{\beta}_i])$ .

**Fairness between flows with same  $RTT$ :** From (6), when flows  $i, j$  have the same  $RTT$ , and so  $\tilde{\alpha}_i = \tilde{\alpha}_j$ , and the same mean backoff factor  $\mathbb{E}[\tilde{\beta}_i] = \mathbb{E}[\tilde{\beta}_j]$  then they obtain on average the same throughput share.

**Fairness between flows with different  $RTTs$ :** When flows  $i, j$  have different round trip times  $T_i \neq T_j$  but the same mean backoff factor, the ratio of their throughputs is  $\mathbb{E}[s_i]/\mathbb{E}[s_j] = (T_j/T_i)^2$ . Observe that this is identical to standard TCP behavior [32].

**Fairness between flows with different loss rates:** The stationary mean backoff factor  $\mathbb{E}[\tilde{\beta}_i]$  depends on the probability that flow  $i$  experiences a packet loss at a network backoff event. Hence, if two flows  $i$  and  $j$  experience different per packet loss rates  $p_i$  and  $p_j$  (e.g. they might have different access links while sharing a common throughput bottleneck), this will affect fairness through  $\mathbb{E}[\tilde{\beta}_i]$ .

**Friendliness:** The model (4) is sufficiently general to include AIMD with fixed backoff factor, as used by standard TCP. We consider two cases. First, when the link is loss-free (the only losses are due to queue overflow) and all flows backoff when the queue fills, then  $1 - \mathbb{E}[\tilde{\beta}_i] = 1 - \beta_i(k)$ . Hence, for a flow  $i$  with fixed backoff of 0.5 and a flow  $j$  with adaptive backoff  $\beta_j$ , when the flows have the same  $RTT$  the ratio of the mean flow throughputs is  $\mathbb{E}[s_i]/\mathbb{E}[s_j] = 2(1 - \beta_j)$  by (6). When  $\beta_j = T_j/RTT_j = 0.5$  then the throughputs are equal. Since  $RTT_j = T_j + q_{max}/B$  where  $q_{max}$  is the link buffer size and  $B$  the link rate, then  $\beta_j = 0.5$  when  $q_{max} = BT_j$ , i.e. the buffer is size at the bandwidth-delay product. The second case we consider here is when the link has i.i.d packet losses with probability  $p$ . When  $p$  is sufficiently large that the queue rarely fills, then queue overflow losses are rare and the throughput

of a flow  $i$  with fixed backoff of 0.5 is accurately modeled by the Padhye model [29]. That is, the throughput is largely decoupled from the behavior of other flows sharing the link (since coupling takes place via queue overflow) and, in particular, this means that flows using adaptive backoff do not penalize flows which use fixed backoff. We present experimental measurements confirming this behavior in Section 6.3.

## 4.4 Network Coding Operations

The coding operations are performed over blocks (Figure 1). Unlike TCP/NC [7], we do not use a sliding window for coding operations. The main reason behind this design decision is for delay and complexity. The sliding window approach allows for better throughput performance; however, when using this approach, the receiver may not be able to decode even the first packet of the file until the entire file is received. As a result, the decoding complexity may be high, as the decoding operation may be performed over the entire file (instead of segments of the file). This may not be a significant concern for small file transfers; however, for some applications such as multimedia streaming and large file transfers, this may be a significant concern. Therefore, in our design, we have opted to use block codes, where we can bound the delay and the complexity by changing the block size *blksize*.

Setting *blksize* = 1 leads to operations similar to that of traditional TCP variants and cannot effectively take advantage of network coding. On the other hand, setting *blksize* too large leads to the problems faced by TCP/NC. Therefore, *blksize* has to be chosen with care. In our experience, it is desirable to set *blksize* to be similar to the bandwidth $\times$ delay of the network. This is because larger block size yields to higher efficiency at the cost of higher delay, and a balance needs to be struck between these competing requirements.

The coding field size has been chosen with some care as well. Similar to block size, using higher field size leads to higher probability of generating independent dofs (leading to efficiency); however, this comes at the cost of coding and decoding complexity. We use a field of  $\mathbb{F}_{256}$ , i.e. each coefficient is a single byte, as it has been shown to provide good performance [7].

To ensure that coding is only performed when necessary, we use systematic block codes – i.e. uncoded packets are transmitted before coded packets are sent. In generating coded packets, there are many options. The sender may only code a subset of the packets in a block. In our design, we use a simple approach – a coded packet is generated by randomly coding all packets in the block. This approach is most effective in terms erasure correction. With high probability, a coded packet will correct for any single erasure in the block.

## 4.5 Transmission and Block Scheduling

```

Initialize an array onfly[] to 0;
for seqno in [seqno_una, seqno_nxt - 1] do
    if current time <  $T(\textit{seqno}) + 1.5RTT$  then
        | onfly[B(seqno)]  $\leftarrow$  onfly[B(seqno)] + 1;
    end
end
for blkno in [currrblk, currrblk + numblks - 1] do
    if blkno = currrblk and
    (1 - p)onfly[currrblk] < blksize - currdof then
        | Transmit a packet with sequence number
        |   seqno_nxt from block blkno;
        |   seqno_nxt  $\leftarrow$  seqno_nxt + 1;
    else if (1 - p)onfly[blkno] < blksize then
        | Transmit a packet with sequence number
        |   seqno_nxt from block blkno;
        |   seqno_nxt  $\leftarrow$  seqno_nxt + 1;
    end
end

```

**Algorithm 3:** CTCP sender algorithm for block scheduling when a token is available.

When a token is available, the CTCP sender decides which block to transmit a packet from. The block scheduling algorithm (Algorithm 3) plays a key role in CTCP's operations. The algorithm first computes the number of packets in transit from the sender to the receiver. Given *p*, the sender can compute the expected number of packets the receiver will receive for any given block. In determining the expected number of dofs the receiver will receive for any given block, we exclude the packets that have been transmitted more than  $1.5 \cdot RTT$  time ago, as they are likely to be lost or significantly delayed. The constant factor of 1.5 may be adjusted depending on the delay constraints of the application of interest; however, the constant factor should be  $\geq 1$ .

The goal of the sender is to ensure that, in expectation, the receiver will receive enough packets to decode the block. The sender prioritizes block *i* before *i* + 1; therefore, *currrblk* is of the highest priority. Note that the algorithm treats *currrblk* slightly differently from the rest of the blocks. In our design, the CTCP receiver informs the sender of how many dofs it has received (*currdof*) for block *currrblk*. Therefore, the sender is able to use the additional information to determine more precisely whether another packet should be sent from block *currrblk* or not. It is not difficult to piggy-back more information on the ACKs. For example, we could include how many dofs the receiver has received for blocks *currrblk* as well as *currrblk* + 1, *currrblk* + 2, ..., *currrblk* + *numblks* - 1. However, for simplicity, the CTCP receiver only informs the sender the number of dofs received for block *currrblk*.

In Algorithm 3, we assume that all blocks are of

```

cc  $\leftarrow$  coding coefficients of the received packet;
pp  $\leftarrow$  payload of the received packet;
index  $\leftarrow$  index of the first non-zero element in cc;
if  $C_{blkno}[\textit{index}, :]$  is empty then
    | pivot  $\leftarrow$  value of c at index index;
    | Insert cc/pivot into  $C_{blkno}[\textit{index}, :]$ ;
    | Insert pp/pivot into  $P_{blkno}[\textit{index}, :]$ ;
    | return TRUE;
else
    if index < blksize then
        | pivot  $\leftarrow$  value of c at index index;
        | cc  $\leftarrow$  cc - pivot ·  $C_{blkno}[\textit{index}, :]$ ;
        | pp  $\leftarrow$  pp - pivot ·  $P_{blkno}[\textit{index}, :]$ ;
        | pivot  $\leftarrow$  value of cc at index index + 1;
        | cc  $\leftarrow$  c/pivot;
        | pp  $\leftarrow$  p/pivot;
        | if cc ≠ 0 then
            | Recursively call itself with updated cc
            |   and pp;
        | end
    end
    return FALSE;
end

```

**Algorithm 4:** CTCP receiver algorithm for updating  $C_{blkno}$  and  $P_{blkno}$  when a packet from block *blkno* is received. We denote *cc* to be the coding coefficients and *pp* the (coded) payload of the received packet.

length *blksize*. We note that CTCP can cope with blocks of varying length; however, for simplicity of presentation, we have chosen to present the algorithms with a fixed block length.

## 5. CTCP RECEIVER

We now present the receiver side algorithm for CTCP. The receiver is responsible for decoding the received data. Another important role of the receiver is to construct ACKs for the sender. Whenever the receiver receives a packet, it needs to check whether the current block is decodable (*ack\_currrblk*) and how many dofs it has received for the current block (*ack\_currdof*).

### 5.1 Decoding Operations

The CTCP receiver also organizes the received packets into blocks. For each block *blkno*, the receiver initializes a *blksize* × *blksize* matrix  $C_{blkno}$  for the coding coefficients and a corresponding payload structure  $P_{blkno}$ . Whenever a packet from *blkno* is received, the coding coefficients and the coded payload are inserted to  $C_{blkno}$  and  $P_{blkno}$  respectively as shown in Algorithm 4. Algorithm 4 returns FALSE if the packet is linearly dependent to the previously received packets; otherwise it returns TRUE. Note that Algorithm 4 ensures that  $C_{blkno}$  is an upper-triangular matrix with diagonal en-

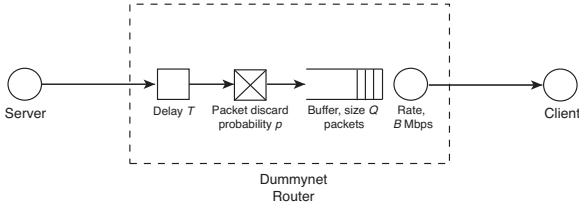


Figure 2: Schematic of experimental testbed.

tries equal to one. Since CTCP sender uses a systematic code, CTCP receiver may often be able to insert  $pp$  and  $cc$  directly – i.e. row  $index$  of  $C_{blkno}$  is empty.

When Algorithm 4 returns TRUE, the receiver sets  $ack\_currdo\!f \leftarrow ack\_currdo\!f + 1$ . If  $ack\_currdo\!f$  is equal to  $blksize$ , then the receiver acknowledges that enough dofs have been received for  $ack\_currblk$  and update  $ack\_currblk \leftarrow ack\_currblk + 1$  ( $ack\_currdo\!f$  is also reset to reflect the dofs needed for the new  $ack\_currblk$ ). If Algorithm 4 returns FALSE, then the receiver transmits an ACK (corresponding to the packet received); however, it does not update  $ack\_currdo\!f$  nor  $ack\_currblk$ .

Once enough dofs are received for a block, the receiver can decode all packets within the block. This results in performing a Gauss-Jordan elimination on an upper-triangular matrix  $C_{blkno}$  and its corresponding  $P_{blkno}$ .

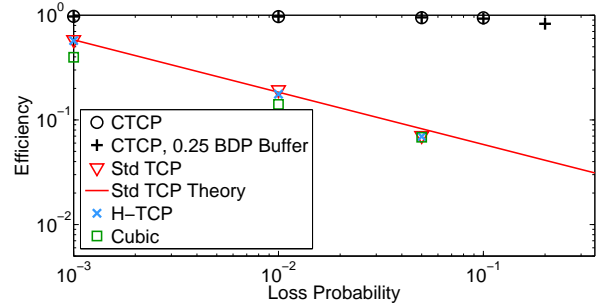
## 6. EXPERIMENTAL MEASUREMENTS

In this section, we demonstrate CTCP’s performance in a testbed. We present results on not only throughput but also on friendliness and fairness.

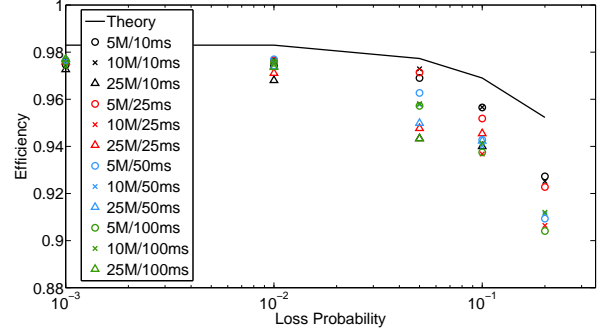
### 6.1 Testbed Setup

The lab testbed consists of commodity servers (Dell Poweredge 850, 3GHz Xeon, Intel 82571EB Gigabit NIC) connected via a router and gigabit switches (Figure 2). Sender and receiver machines used in the tests both run a Linux 2.6.32.27 kernel. The router is also a commodity server running FreeBSD 4.11 and `ipfw-dummynet`. It can be configured with various propagation delays  $T$ , packet loss rates  $p$ , queue sizes  $Q$  and link rates  $B$  to emulate a range of network conditions. As indicated in Figure 2, packet losses in `dummynet` occur before the rate constraint, not after, and so do not reduce the bottleneck link capacity  $B$ . Unless otherwise stated, appropriate byte counting is enabled for standard TCP and experiments are run for at least 300s. Data traffic is generated using `rsync` (version 3.0.4), HTTP traffic using `apache2` (version 2.2.8) and `wget` (version 1.10.2), video traffic using `vlc` as both server and client (version 0.8.6e as server, version 2.0.4 as client).

CTCP is implemented in userspace as a forward proxy located on the client and a reverse proxy located on the server. This has the advantage of portability and of requiring neither root-level access nor kernel changes. Traffic between the proxies is sent using CTCP. With this setup, a client request is first directed to the local



(a) Link 25Mbps, RTT 20ms



(b) CTCP

Figure 3: Measurements of goodput efficiency against packet loss rate, link rate and RTT. The Theory curve in Figure 3b is generated using Equation (7).

forward proxy. This transmits the request to the reverse proxy, which then sends the request to the appropriate port on the server. The server response follows the reverse process. The proxies support the SOCKS protocol and standard tools allow traffic to be transparently redirected via the proxies. In our tests, we used `proxychains` (version 3.1) for this purpose.

### 6.2 Efficiency

Figure 3 presents experimental measurements of the efficiency (equal to  $\frac{\text{goodput}}{\text{link capacity}}$ ) of standard TCP and CTCP over a range of network conditions. Figure 3a shows the measured efficiency versus the packet loss probability  $p$  for a 25Mbps link with 25ms RTT and a bandwidth-delay product of buffering. Baseline data is shown for standard TCP (i.e. TCP SACK/Reno), Cubic TCP (current default on most Linux distributions), H-TCP, together with the value  $\sqrt{1.5/p}$  packets per RTT predicted by the popular Padhye model [29]. It can be seen that the measurements for standard TCP are in good agreement with the Padhye model, as expected. Also that Cubic TCP and H-TCP closely follow standard TCP behavior, again as expected since the link bandwidth-delay product of 52 packets lies in the regime where these TCP variants seek to ensure backward compatibility with standard TCP. Observe that the achieved goodput decreases rapidly with increasing loss rate, falling to 20% of the link capacity when the



packet loss rate is 1%. This feature of standard TCP is, of course, well known. Compare this, however, with the efficiency measurements for CTCP, which are shown in Figure 3a and also given in more detail in Figure 3b. The goodput is  $> 96\%$  of link capacity for a loss rate of 1%, a roughly five-fold increase in goodput compared to standard TCP.

Figure 3b presents the efficiency of CTCP for a range of link rates, RTTs and loss rates. It shows that the efficiency achieved is not sensitive to the link rate or RTT. Also shown in Figure 3b is a theoretical upper bound on the efficiency calculated using

$$\eta = \frac{1}{N} \sum_{k=0}^{n-1} (n-k) \binom{n}{k} p^k (1-p)^{N-k}, \quad (7)$$

where  $N = 32$  is the block size,  $p$  the packet erasure probability and  $n = \lfloor N/(1-p) \rfloor - N$  is the number of forward-transmitted coded packets sent with each block. This value  $\eta$  is the mean number of such forward-transmitted coded packets that are unnecessary (because there are fewer than  $n$  erasures).

The efficiency achieved by CTCP is also insensitive to the buffer provisioning, as discussed in Section 4.3. This property is illustrated in Figure 3a, which presents CTCP measurements when the link buffer is reduced in size to 25% of the bandwidth-delay product. The efficiency achieved with 25% buffering is close to that with a full bandwidth-delay product of buffering.

### 6.3 Friendliness with Standard TCP

Figures 4 and 5 confirm that standard TCP and CTCP can coexist in a well-behaved manner. In these measurements, a standard TCP flow and a CTCP flow share the same link competing for bandwidth. As a baseline, Figure 4 presents the goodputs of TCP and CTCP for range of RTTs and link rates on a *loss-free* link (i.e. when queue overflow is the only source of packet loss). As expected, it can be seen that the standard TCP and CTCP flows consistently obtain similar goodput.

Figure 5 presents goodput data when the link is lossy. The solid lines indicate the goodputs achieved by the CTCP flow and the standard TCP flow sharing the same link with varying packet loss rates. At low loss rates, they obtain similar goodputs; but as the loss rate increases, the goodput of standard TCP rapidly decreases (as already observed in Figure 3a).

For comparison, in Figure 5, we also show (using the dotted lines) the goodput achieved by a standard TCP flow when competing against another standard TCP flow (i.e. when two standard TCP flows share the link). Note that the goodput achieved by a standard TCP flow (dotted line) when competing against another standard TCP flow is close to that achieved when sharing the link with a CTCP flow (solid line). This demonstrates that CTCP does not penalize the standard TCP flow.

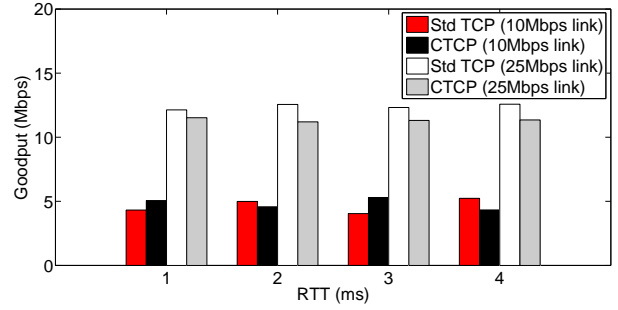
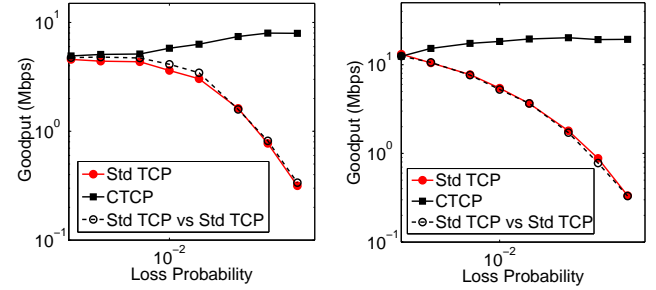


Figure 4: Goodput for a standard TCP and a CTCP flow sharing a loss-free link; results are shown for 10Mbps and 25Mbps links with varying RTTs.



(a) Lossy 10Mbps link with RTT=25ms (b) Lossy 25Mbps link with RTT=25ms

Figure 5: Goodput against link loss rate for (i) a TCP and a CTCP flow sharing this link (solid lines), and (ii) two TCP flows sharing lossy link (dashed line).

### 6.4 Fairness among CTCP Flows

We turn now to fairness, i.e. how goodput is allocated between competing CTCP flows. Figure 6a plots a typical *tokens* time history for two CTCP flows sharing a lossy link. The second flow (black) is started after the first (grey) so that we can observe the convergence to fairness. It can be seen that the two flows' *tokens* rapidly converge. Figure 6b presents corresponding goodput measurements for a range of link rates, RTTs, and loss rates. Again, the two CTCP flows consistently achieve similar goodputs.

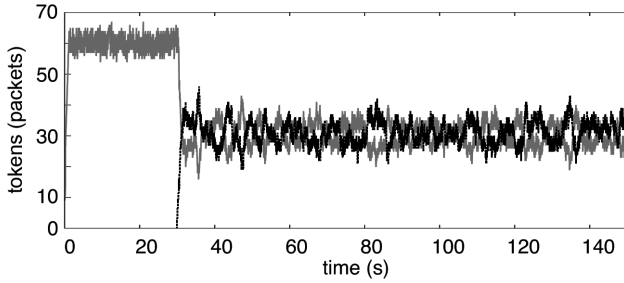
### 6.5 Application Performance

In this section, we present our testbed results seen by various applications.

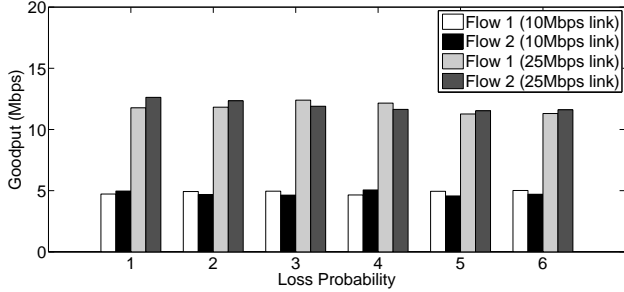
#### 6.5.1 Web

Figure 7 shows measurements of HTTP request completion time against file size for standard TCP and CTCP. The HTTP requests are generated using `wget` and the response is by an `apache2` web server. Note the log scale on the y-axis.

The completion times with CTCP are largely insensitive to the packet loss rate. For larger file sizes, the completion times approach the best possible performance



(a) 25Mbps, RTT 25ms, 5% packet loss rate



(b) RTT 25ms

Figure 6: Measurements of *tokens* and goodput for two CTCP flows sharing a lossy link. Figure 6a provides a sample of *tokens* time history while Figure 6b summarizes the goodputs for a range of packet loss rates (%) with 10Mbps and 25Mbps links (RTT 25ms). Similar behavior was observed for other values of RTTs.

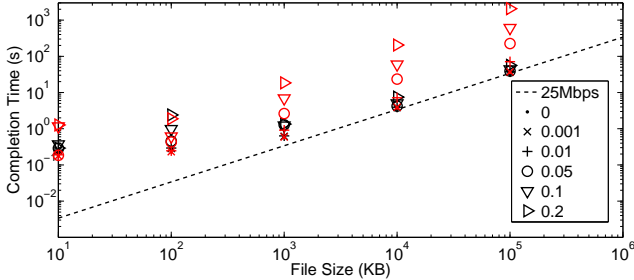


Figure 7: Measured HTTP request mean completion time against file size over 25Mbps link with RTT = 10ms. Data is shown for standard TCP (red) and CTCP (black) for a range of loss rates. Error bars are comparable in size to the symbols used in the plot and so are omitted.

indicated by the dashed line. For smaller file sizes, the completion time is dominated by slow-start behavior. Note that CTCP and TCP achieve similar performance when the link is loss-free; however, TCP's completion time quickly increases with loss rate. For a 1MB connection, the completion time with standard TCP increases from 0.9s to 18.5s as the loss rate increases from 1% to 20%, while for a 10MB connection the corresponding increase is from 7.1s to 205s.

### 6.5.2 Streaming Video

Figure 8 plots performance measurements for stream-

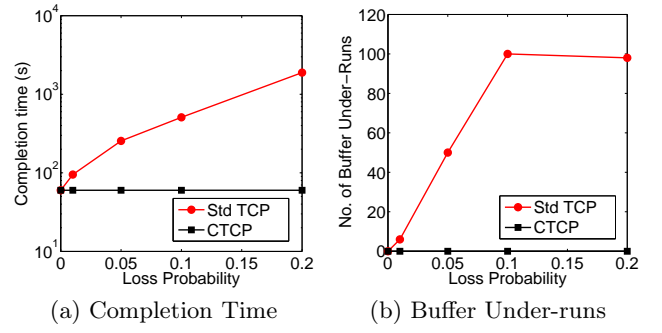


Figure 8: Measurements of video streaming performance against loss rate with a 25Mbps link and a RTT of 10ms. Data is shown for standard TCP and CTCP. Figure 8a shows the running time taken to play a video of nominal duration (60s); Figure 8b shows the number of under-runs of the playback buffer at the client.

ing video for a range of packet loss rates on a 25Mbps link with RTT equal to 10 ms. A vlc server and client are used to stream a 60s video. Figure 8a plots the measured time for playout of the video to complete. Again, note the log scale on the y-axis.

The playout time with CTCP is approximately 60s and is insensitive to the packet loss rate. In contrast, the playout time with standard TCP increases from 60s to 95s when the loss rate is increased from 0% to 1%, and increases further to 1886s (31 minutes) as the loss rate is increased to 20%. Figure 8b plots measurements of playout buffer under-run events at the video client. It can be seen that there are no buffer under-run events when using CTCP even when the loss rate is as high as 20%. With standard TCP, the number of buffer under-runs increases with loss rate until it reaches a plateau at around 100 events, corresponding to a buffer underrun occurring after every playout of a block of frames. In terms of user experience, the increases in running time result in the video repeatedly stalling for long periods of time and are indicative of a unsatisfactory quality of experience even at a loss rate of 1%.

## 7. REAL-WORLD PERFORMANCE

In this section, the performance of CTCP and TCP are measured in production networks to determine real-world gains of the new protocol. Specifically, experiments were executed on both an IEEE 802.16 cellular network [33] and several public WiFi networks.

### 7.1 IEEE 802.16 WiMAX Network

#### 7.1.1 Experiment Setup

The IEEE 802.16 WiMAX network used was made available through the Global Environment for Network Innovations (GENI) collaborative research framework [34]. Experiments were run using a single WiMAX base

Table 2: IEEE 802.16 BS and SS Configurations.

BS		SS	
Tx. Power	CINR	RSSI	Tx. Power
21 dBm	20 dB	-71 dBm	-63 dBm
22 dBm	21 dB	-70 dBm	-63 dBm
23 dBm	22 dB	-69 dBm	-63 dBm
24 dBm	23 dB	-68 dBm	-63 dBm
25 dBm	23 dB	-68 dBm	-63 dBm
26 dBm	24 dB	-67 dBm	-63 dBm

station (BS) acting as the server and a single WiMAX subscriber station (SS) acting as the client (both running a Ubuntu 10.10 kernel). The downlink modulation and coding scheme (MCS) for every experiment remained constant (64-QAM CTC  $5/6$ ), while the transmit power levels were adjusted to obtain different packet loss rates and simulate varying distances between the WiMAX BS and SS. The BS transmit power, and corresponding SS signal to noise ratios (SNR), were chosen to reflect those that would be observed in deployed cellular networks. While increasing the BS transmit power will result in fewer losses, it would not necessarily reflect a typical cellular network experience.

The average transmit (Tx) power, Carrier to Interference plus Noise Ratio (CINR), Received Signal Strength Indication (RSSI), and MCS used are shown in Table 2. The uplink MCS and Tx power was held constant at 64-QAM, CTC  $1/2$  and -63 dBm respectively. Automatic Repeated reQuest (ARQ) and Hybrid-ARQ (HARQ) are both turned off to avoid any effects on the throughput and delay as a result of these algorithms. A comprehensive study of the interaction between HARQ and ARQ in 802.16 networks with coding is provided in [35].

The same version of CTCP as in Section 6 was used; however, we introduce two changes. First, instead of using a series of proxies to redirect traffic, a simple `ftp` program was generated to send data over CTCP. Second, a modification to the congestion control mechanism was made for some of the experiments so that  $RTT_{min}$  is reset to the last  $RTT$  measurement taken if  $token < 8$ . The reason for this change will be explained in the next subsection. All data, generated using `iperf` (version 2.0.4), was sent using the operating system's default TCP implementation (i.e., Cubic TCP). Experiments were run for each of the BS Tx power levels indicated in Table 2, standard TCP and CTCP data transfers were executed individually without any additional traffic on the network, and all experiments were run for a minimum of 300 s.

### 7.1.2 Results and Discussion

The WiMAX network provides insight into the performance of CTCP in a production cellular network. Specifically, these networks experience a wide range of

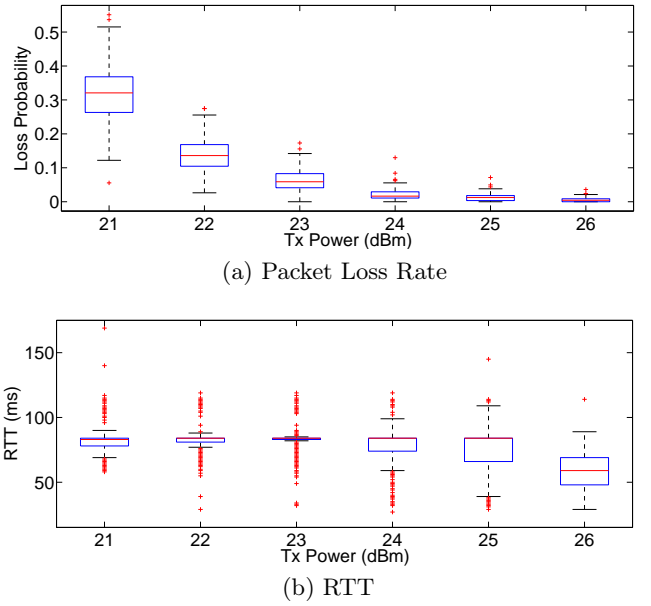


Figure 9: Measured WiMAX sample distributions for packet loss rate and RTT. The center line represents the median, and the top/bottom of the box represents the 75th/25th percentiles respectively. The whiskers extend to the most extreme data points not considered outliers and outliers are plotted individually.

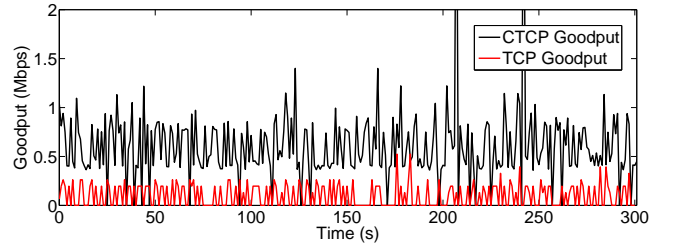


Figure 10: TCP's and the modified CTCP's goodput with a WiMAX BS Tx power of 21 dBm.

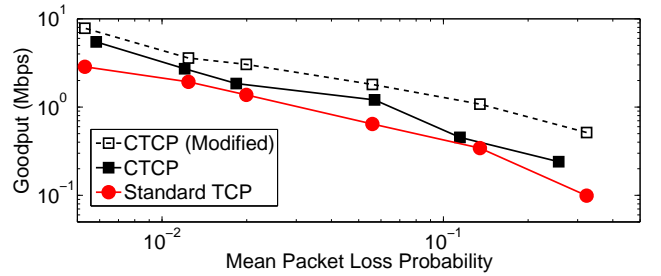


Figure 11: Mean goodput as a function of the mean packet loss probability for the WiMAX network.

packet loss rates and RTT jitter for any given SNR. Figure 9 provides the distribution of packet loss rate and RTT measured.

A sample of the results is shown in Figure 10. The BS Tx power was set to 21 dBm. The black line is the measured goodput for the modified CTCP version, and

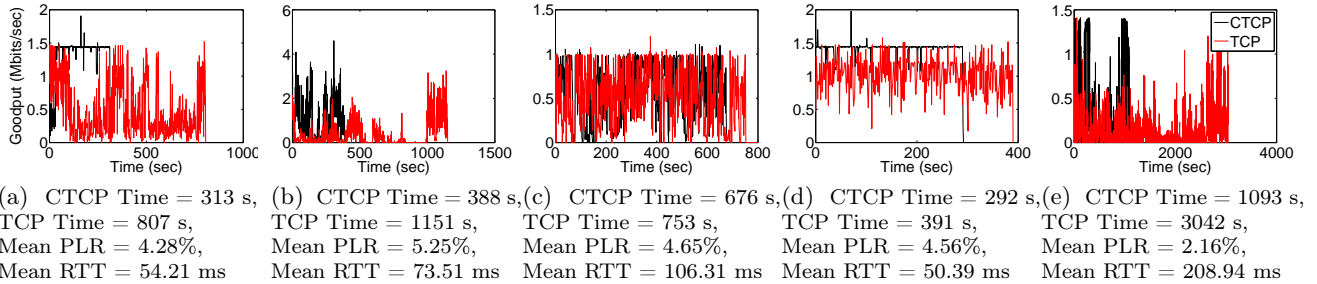


Figure 12: Public WiFi Network Test Traces (CTCP in black, TCP in red). The download completion times, the mean packet loss rate (*PLR*), and mean *RTT* for each experiment are also provided.

the red line is the standard TCP goodput. The average packet loss rate for the trace shown is 0.3238. The mean standard TCP goodput and mean CTCP goodput are 0.09917 Mbps and 0.5148 Mbps, respectively; resulting in a gain of approximately 5 times that of standard TCP. In addition, CTCP's mean efficiency was 0.9406, which is consistent with the results presented earlier. Finally, the mean goodput measurements for all BS Tx power settings are shown in Figure 11.

These figures show several interesting results. First, the spikes in Figure 10 at approximately 200 and 250 seconds indicate CTCP decoding events. Since the loss rate is approximately 0.3 when Tx power is 21 dBm, CTCP introduces redundant packets. Prior to these spikes, the number of blocks (therefore, packets) that cannot be decoded is fairly large. Upon receipt of enough dofs from the server, the client can decode and pass a large number of packets up to the application layer, resulting in the spikes shown in Figure 10.

Second, the unmodified CTCP gains with respect to standard TCP, shown in Figure 11, are not nearly as large as those shown in Figure 5. The gain is reduced because of the RTT jitter rather than the capacity of the network (the network capacity has been measured to be greater than 10 Mbps for all Tx power settings). As shown in Figure 9, the majority of measured RTT samples are approximately 84 ms for Tx powers less than 25 dBm, while the sample values ranged widely from 26 ms to 264 ms. This variability is not necessarily a result of congestion but rather the 802.16 MAC. When an RTT sample less than 84 ms is measured, CTCP's  $RTT_{min}$  is updated to a value that is not necessarily the base RTT, which should represent the true path delay. Because  $tokens$  is reduced by  $\frac{RTT_{min}}{RTT}$  every time a packet is lost,  $tokens$  reduced to its minimum quickly under high loss rates resulting in a significantly lower throughput. In order to counteract the effects of RTT jitter on CTCP, a small modification to the congestion control algorithm was introduced specifically for the WiMAX tests. This change resets  $RTT_{min}$  to the last  $RTT$  measurement when  $tokens$  drops below the

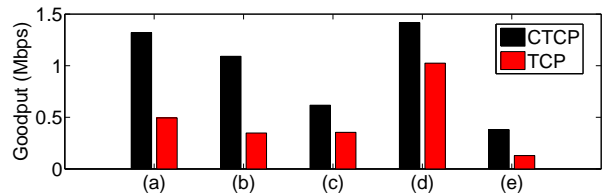


Figure 13: Mean goodput for each of the experiments shown in Figure 12.

initial  $tokens$  value (i.e.,  $tokens < 8$ ). This approach does not necessarily increase the goodput of short lived sessions, but does benefit the goodput of longer sessions by approximately twice that of the unmodified CTCP.

When comparing the gains we observed with those presented in [35], we find that introducing network coding at the transport layer alone is not necessarily the best strategy for increasing throughput; but it does offer a level of flexibility that cannot be achieved by [35]. In [35], network coding was introduced just above the MAC layer and was shown to provide throughput gains of up to 5.9 (for UDP traffic). Their implementation aims to improve throughput over a single WiMAX link and requires changes to the WiMAX BS, which does not provide a complete end-to-end solution and is more difficult to deploy. However, our results, as well as [35], indicate that including network coding throughout the network stack may further increase throughput.

## 7.2 Public WiFi Network

In addition to WiMAX, we conducted several experiments over public WiFi networks. A 50 MB file was downloaded from a server (running Ubuntu 10.04.3 LTS) located on the MIT campus to a laptop (running Ubuntu 12.04.1 LTS) using various public WiFi networks in the greater Boston area. We use the simple `ftp` program described in Section 7.1 to generate and send data over CTCP, and Linux's native secure copy program (`scp`) to test the standard TCP performance.

Figure 12 shows the traces for five of the experiments. It is important to point out that standard TCP stalled and had to be restarted twice before successfully com-

pleting in the test shown in Figure 12c. CTCP, on the other hand, never stalled nor required a restart.

Each trace represents a different WiFi network that was chosen because of the location, accessibility, and perceived congestion. For example, the experiments were run over WiFi networks in shopping center food courts, coffee shops, and hotel lobbies. In Figures 12a - 12d, the WiFi network spanned a large user area increasing the possibility of hidden terminals; a scan of most of the networks showed  $> 40$  active WiFi radios, which also increases the probability of collisions. The only experiment that had a small number of terminals (i.e. five active radios) is shown in Figure 12e.

In each of the experiments, CTCP achieved a larger average goodput and faster completion time. The average throughput for both CTCP and TCP is shown in Figure 13. Taking the mean throughput over all of the conducted experiments, CTCP achieves a goodput of approximately 750 kbps while standard TCP achieves approximately 300 kbps; resulting in a gain of approximately 2.5. If we take into account the mean packet loss rate of approximately 4%, the gain presented here is similar to that of the WiMAX experiments presented in Section 7.1. Finally, we note that in several of the experiments, CTCP experienced fewer timeouts and was able to maintain higher throughput than TCP.

We emphasize the observed loss rates of approximately 4% in Figure 12, which is quite high and unexpected; resulting in CTCP's significant performance gain over TCP's. We believe that the loss rate is not only due to randomness but also due to congestion, interference, and hidden terminals. This is an area that would be worthwhile to investigate further. If our intuition is indeed correct, we believe that CTCP can greatly help increase efficiency in challenged network environments.

## 8. CONCLUSIONS AND FUTURE WORK

We proposed CTCP, a reliable transport protocol that uses network coding. We implemented CTCP within the application layer (over UDP). As a result, our implementation requires neither kernel level modifications nor changes to the network's internal nodes, which may allow CTCP to be more easily deployed. We further showed that CTCP performs significantly better than standard TCP, especially in lossy networks.

There are areas for further research. CTCP's congestion control mechanism performs well in lossy networks where an increase in RTT indicates congestion. When the RTT also varies with non-congestion events such as variability resulting from specific MAC implementations, as shown in 7.1, CTCP's congestion control can needlessly limit throughput. New approaches for fairness and friendliness may be needed for such networks. In addition, we did not investigate the potential impact of active queue management (AQM) on CTCP. How-

ever, the effect of AQM may not be significant as fewer networks use AQMs with the introduction of protocols using streamlets, selective repeat mechanisms, and new congestion control mechanisms such as Cubic. Another possible extension is to allow re-encoding within the network [8, 26, 36], although this may require changes within the network (not just end-to-end). However, this approach has shown to increase efficiency. Finally, we believe that CTCP can be extended to provide gains in multi-path environments [37]. By coding over multiple paths, initial simulations and experiments in [37] show that we can achieve the sum rate of each path without the complexity of tracking and scheduling individual packets over each path.

**Acknowledgements:** The authors would like to thank Ivan Seskar from WINLAB at Rutgers University for his support during the WiMAX experimentation. This material is based upon work supported by DARPA and Space and Naval Warfare Systems Center Pacific under contract no. N66001-11-C-4003.

## 9. REFERENCES

- [1] J. Sommers and P. Barford, "Cell vs. WiFi: on the performance of metro area mobile connections," in *Proceedings of ACM IMC*, pp. 301–314, 2012.
- [2] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei, "Network performance of smart mobile handhelds in a university campus wifi network," in *Proceedings of ACM IMC*, pp. 315–328, 2012.
- [3] M. Dischinger, A. Haeberlen, K. Gummadi, and S. Saroiu, "Characterizing residential broadband networks," in *Proceedings of ACM SIGCOMM*, 2007.
- [4] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: a fresh look at TCP in the wild," in *Proceedings of ACM SIGCOMM*, pp. 76–89, 2009.
- [5] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, pp. 1204–1216, 2000.
- [6] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 782–795, 2003.
- [7] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, "Network coding meets TCP: Theory and implementation," *Proceedings of IEEE*, vol. 99, pp. 490–512, March 2011.
- [8] T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Trans. Inf. Theory*, vol. 52, pp. 4413–4430, October 2006.
- [9] M. Kim, M. Médard, and J. Barros, "Modeling network coded TCP throughput: A simple model



- and its validation,” in *Proceedings of ICST/ACM Valuetools*, May 2011.
- [10] C. Barakat and A. A. Fawal, “Analysis of link-level hybrid FEC/ARQ-SR for wireless links and long-lived tcp traffic,” in *Proceedings of IEEE WiOpt*, 2003.
  - [11] D. Barman, I. Matta, E. Altman, and R. Azouzi, “TCP optimization through FEC, ARQ and transmission power tradeoffs,” tech. rep., Boston University, 2003.
  - [12] C. Barakat and E. Altman, “Bandwidth tradeoff between TCP and link-level FEC,” *Computer Networks*, vol. 39, pp. 133–150, June 2002.
  - [13] B. Liu, L. Dennis, A. Goeckel, and D. Towsley, “TCP-cognizant adaptive forward error correction in wireless networks,” in *Proceedings of IEEE GLOBECOM*, 2002.
  - [14] A. Chockalingam, M. Zorzi, and V. Tralli, “Wireless TCP performance with link layer FEC/ARQ,” in *Proceedings of IEEE ICC*, 1999.
  - [15] D. Kliazovich, M. Bendazzolo, and F. Granelli, “TCP-aware forward error correction for wireless networks,” in *Proceedings of ICST Mobilight*, 2010.
  - [16] M. Miyoshi, M. Sugano, and M. Murata, “Performance improvement of TCP on wireless cellular networks by adaptive FEC combined with explicit loss notification,” in *Proceedings of IEEE Vehicular Technology Conference*, 2002.
  - [17] I. Ahmad, D. Habibi, and M. Z. Rahman, “An improved FEC scheme for mobile wireless communication at vehicular speeds,” in *Proceedings of ATNAC*, 2008.
  - [18] T. Tsugawa, N. Fujita, T. Hama, H. Shimonishi, and T. Murase, “TCP-AFEC: An adaptive FEC code control for end-to-end bandwidth guarantee,” *Packet Video*, 2007.
  - [19] T. Porter and X.-H. Peng, “Effective video content distribution by combining TCP with adaptive FEC coding,” in *Proceedings of IEEE BMSB*, 2010.
  - [20] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. K. Ramakrishnan, “LT-TCP: End-to-end framework to improve TCP performance over networks with lossy channels,” in *Proceedings of IEEE IWQoS*, 2005.
  - [21] L. Baldantoni, H. Lundqvist, and G. Karlsson, “Adaptive end-to-end FEC for improving TCP performance over wireless links,” in *Proceedings of IEEE ICC*, 2004.
  - [22] T. Anker, R. Cohen, and D. Dolev, “Transport layer end-to-end error correcting,” tech. rep., Hebrew University, Leibniz Center, 2004.
  - [23] H. Lundqvist and G. Karlsson, “TCP with end-to-end FEC,” in *International Zurich Seminar on Communications*, 2004.
  - [24] C. Padhye, K. J. Christensen, , and W. Moreno, “A new adaptive FEC loss control algorithm for voice over IP applications,” in *Proceedings of IPCCC 2000*, 2000.
  - [25] V. Subramanian, “Transport and link-level protocols for wireless networks and extreme environments.” Ph.D. Thesis, RPI.
  - [26] D. S. Lun, M. Médard, R. Koetter, and M. Effros, “On coding for reliable communication over packet networkst,” *Physical Communication*, vol. 1, pp. 3–20, March 2008.
  - [27] S. Chachulski, M. Jennings, S. Katti, and D. Katabi, “Trading structure for randomness in wireless opportunistic routing,” in *Proceedings of ACM SIGCOMM*, 2007.
  - [28] P. U. Tournoux, “Un protocole de fiabilité basé sur un code à effacement“on-the-fly”.” Ph.D. Thesis, Université de Toulouse.
  - [29] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, “Modeling TCP Reno performance: a simple model and its empirical validation,” *IEEE/ACM Trans. Netw.*, vol. 8, pp. 133–145, 2000.
  - [30] K. Tan, J. Song, Q. Zhang, and M. Sridharan, “A compound TCP approach for high-speed and long distance networks,” in *Proceedings of INFOCOM*, 2006.
  - [31] R. N. Shorten and D. J. Leith, “On queue provisioning, network efficiency and the transmission control protocol,” *IEEE/ACM Trans. Netw.*, vol. 15, pp. 866–877, 2007.
  - [32] R. Shorten, F. Wirth, and D. Leith, “A positive systems model of TCP-like congestion control: asymptotic results,” *IEEE/ACM Trans. Netw.*, vol. 14, pp. 616–629, 2006.
  - [33] IEEE 802.16 Working Group, “IEEE 802.16: Broadband Wireless Metropolitan Area Networks (MANS).”
  - [34] “Global Environment for Network Innovations (GENI).” <http://www.geni.org>.
  - [35] S. Teerapittayanon, K. Fouli, M. Médard, M. Montpetit, X. Shi, I. Seskar, and A. Gosain, “Network Coding as a WiMAX Link Reliability Mechanism,” in *Int’l Workshop on Multiple Access Communications*, 2012.
  - [36] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, “Xors in the air: Practical wireless network coding,” in *Proceedings of ACM SIGCOMM*, 2006.
  - [37] M. Kim, A. ParandehGheibi, L. Urbina, and M. Médard, “CTCP: Coded TCP using multiple paths,” in ArXiv <http://arxiv.org/abs/1212.1929>, 2012.